*DRAFT 1.0 – **not** authorized for external release*

Automated Testing Techniques: Design of Automated Testing Environment June 30th 1999

Second Deliverable (FY 1999, Quarter 3) of
Automated Testing Techniques
Jet Propulsion Lab Center Initiative UPN 323-08-5N
Project Officer: Siamak Yassini

Prepared by:
Martin S. Feather
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena CA 91109 USA
email: Martin.S.Feather@jpl.nasa.gov
tel: +1 818 354 1194

EXECUTIVE OVERVIEW

The previous deliverable identified translation as the key to automating test oracle generation. This document presents the design of such a translation component. In the DS-1 RAX work that preceded this center initiative, use of a *procedural* translator raised challenges of creation, understanding, revision and enhancement. This design seeks to circumvent these problems.

1 CONTEXT

Figure 1 (repeated from [Feather & Smith 1999]) shows the architectural context of the translator. The inputs to the translator are constraints and type checking rules expressed in the planner-specific notation. Outputs from the translator are database queries which, when executed by the database engine against a database loaded with plan data, will determine whether those constraints and rules hold of that plan. The two notations (planner-specific, and database specific) differ significantly, and the primary purpose of the translator is to bridge this language gap.

In practice, there is need for more than simply a

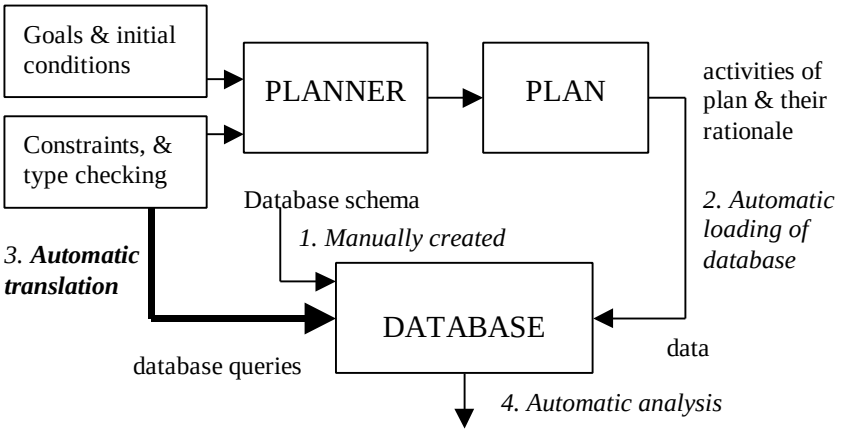

Figure 1 – Planchecker Architecture

yes/no answer to the question of whether all the constraints and rules hold of a plan. There are three aspects to the more detailed information that is required for this kind of plan checking:

- Further subdividing the overall yes/no answer into several answers, one for each constraint/rule, and, at an even more detail, one for each location within the plan where a constraint/rule is applicable.
- Providing details on *how* a particular constraint was satisfied, or not, as the case may be.
- Further subdividing the assumptions under which the answer applies (e.g., the plan satisfies this constraint provided that the immediately following plan satisfies the following condition…).

It is the simultaneous accommodation of these concerns that makes the automated generation of a plan checker test oracle a non-trivial activity. The bulk of work on generation of test oracles does not take these concerns into account.

2 REQUIREMENTS

The requirements for this design are as follows:

- Automation – the translation must be fully automatic.
- Speed – the translator must be tolerably efficient in its task of converting constraints and type checking rules into the analysis code. (The current procedurally coded translator takes on the order of 10 minutes to translate the entire DS-1 RAX constraint set).
- Understandability – the translation logic should be in a form that spacecraft experts can inspect and review.
- Modifiability/Extensibility – the translator should admit to relatively easy maintenance.

The procedurally coded translator that was developed and applied in the course of RAX plan checking met the first two requirements, but fell short of the last two.

3 TRANSLATOR DESIGN ARCHITECTURE

The overall architecture is sketched in Figure 2. The translation from constraints and type checking rules into database queries and the scaffolding code that surrounds them is to be accomplished by a *transformation system.*

Transformation systems have been an active area of research since the 1970s. For surveys of the early days, see [Partsch & Steinbruggen 1983; Feather 1986]. Research in this area continues, for example, a workshop took place in conjunction with this year's International Conference on Software Engineering [STS workshop, 1999], while commercial support for and application of transformation is increasing, for

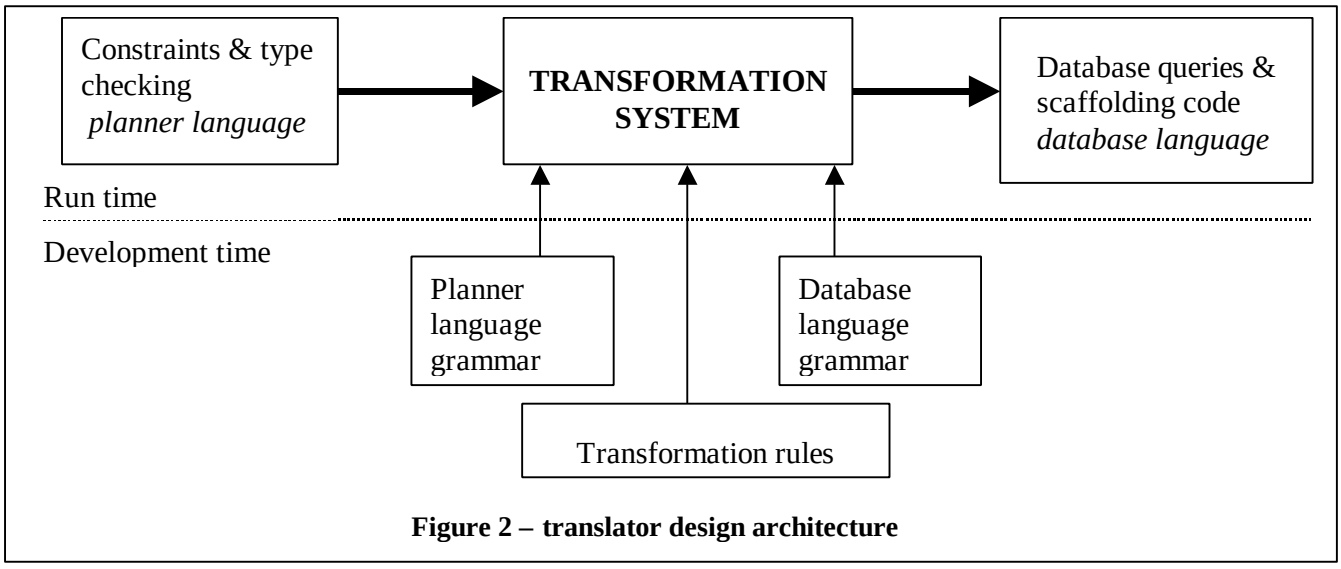

Figure 2 – translator design architecture

example [Reasoning SDK™].

The advantage of this design is that transformation systems separate the transformations themselves from their execution. Typically, a generic transformation engine performs the execution, driven by four inputs. At development time, the developer provides grammars of the input and output languages of the transformation and a set of *declarative transformation rules* that direct the translation process. At run time, the object to be transformed (expressed in the input grammar) is input, and the transformation engine applies the transformation rules to effect the transformation. This separation of the transformation rules from the engine that applies them is key. It encourages and facilitates a more declarative style of expression, with consequent advantages of understandability and maintainability.

In our case, the transformation purpose is to translate between languages. The input language is the planner-specific language for expression of constraints and type checking rules. The output language is the analysis-specific language for checking the adherence of the plan to those constraints and rules. We are using a database as the analysis engine, so the database query language is at the heart of the required form of output. In practice, a significant amount of "scaffolding" code must accompany the database queries (e.g., to direct report generation).

Specifically:

- The plans we are dealing with are written in the language of the HSTS planner as used in DS-1's Remote Agent Experiment [RAX 1999].
- The database we are using for analysis is AP5 [Cohen 1989] a research-quality advanced database tool developed at the University of Southern California.
- The output grammar is this Lisp, augmented by AP5's query capabilities (akin to first-order predicate logic over a relational database).

4 DESIGN INSTANTIATION

The transformation system we plan to use is POPART [Wile 1999]. POPART is well suited to prototyping "Domain Specific Languages", into which category the planner language and the AP5 query language squarely fall.

POPART is representative of the rule-based transformation systems that employ declarative, textual expressions of the transformation rules themselves. Fortuitously, POPART implemented on top of the same language base (Lisp) as the AP5 analysis engine. This makes development and integration of the translator a simpler process than would be the case if disparate languages and language environments were involved.

5 FULFILMENT OF REQUIREMENTS

The identified requirements will be fulfilled by this design as follows:

- Automation – once primed (at development time) with grammars and transformation rules, POPART's run-time transformation will accomplish the translation completely automatically.
- Speed – the switch from a procedurally coded translator to using POPART's translation engine primed with rules may incur some degradation of performance. Note, however, that re-translation of the constraint set is a relatively infrequent operation (in the DS-1 development, there were four releases of the constraint set, spread over the course of several months). A modest degree of performance degradation can therefore be tolerated.
- Understandability – the declarative form of transformation rules has long been recognized as a virtue from the viewpoint of understandability. [Reyes & Richardson, 1998] use declarative transformation to translate from formal specifications to test oracles of the implementations of those specifications. We take this as encouraging evidence of the viability of this approach in the testing arena.
- Modifiability/Extensibility – the procedural code was difficult to modify or extend. In contrast, the declarative rules by their very nature encapsulate the translation concerns more effectively and succinctly. As a result, all forms of maintenance will be facilitated by this design.

6 ALTERNATIVE DESIGN INSTANTIATION

POPART, like the vast majority of transformation systems, uses primarily textual representations of transformation rules and their organization.

An alternative style of translation is to structure and view the overall translator organization as a dataflow-like diagram. This is the approach under investigation in a concurrent Center Initiative, "Automatic Software Code Generator For Spacecraft Fault Protection & Monitors", Task Lead - Nicolas Rouquette.

It would be revealing to implement our particular translation task in both design alternatives (textual and dataflow), so as to emerge with the best result. Likely we will not have the time to do this completely, but experiments with fragments of the overall translation might be both feasible and illuminating.

REFERENCES

[Cohen 1989] D. Cohen. Compiling Complex Database Transition Triggers. *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (Portland, Oregon, 1989), ACM Press, 225-234.

[Feather 1986] M.S. Feather "A survey and classification of some proram transformation approaches and techniques," *Proceedings of the IFIP TC2/WG2.1 Working Conference on Program Specification and Transformation*, L.G.L.T. Meertens (ed), North Holland, 1996: pp. 165-195.

[Feather & Smith 1999] M.S. Feather & B. Smith "Automatic Generation of Test Oracles - From Pilot Studies to Application," *in submission*.

[Partsch & Steinbruggen 1983] H.Partsch & R. Steinbruggen "Program transformation systsems," *ACM Computing Surveys*, 15, 1993 pp. 199-236.

[RAX 1999] http://rax.arc.nasa.gov/

[Reasoning SDK™] Refine User Guide, DIALECT User Guide, Reasoning Systems, Mountain View, CA.

[Reyes & Richardson, 1998] "Specification-based testing of Ada units with low encapsulation," *Proceedings, 13th IEEE International Conference on Automated Software Engineering*, 1998, pp. 22-31.

[STS workshop, 1999] *Proceedings, Software Transformation Systems*, Los Angeles, CA, May 1999.

[Wile 1999]. Popart: Producer of Parsers and Related Tools. http://www.isi.edu/software-sciences/wile/Popart/popart.html